Chapter 10

Fun with Functions

In This Chapter

- Creating a function
- Avoiding the prototype
- Working with variables in a function
- Passing arguments to a function
- Returning values from a function
- Using `return` to leave a function

When it comes getting work done, it's a program's functions that do the heavy lifting. The C language comes with libraries full of functions, which help bolster the basics of the language, the keywords, the operators, and so on. When these C library functions fall short, you concoct your own functions.

Anatomy of a Function

The tools that are needed to craft your own functions are brief. After deciding the function's purpose, you give it a unique name, toss in some parentheses and curly brackets, and you're pretty much done. Of course, the reality is a bit more involved, but it's nothing beyond what I cover in the first two parts of this book.

Constructing a function

All functions are dubbed with a name, which must be unique; no two functions can have the same name, nor can a function have the same name as a keyword.

The name is followed by parentheses, which are then followed by a set of curly brackets. So at its simplest construction, a function looks like this:

```
type function() { }
```

In the preceding line, *type* defines the value returned or generated by a function. Options for *type* include all the standard C variable types — `char`, `int`, `float`, `double` — and also `void` for cheap functions that don't return anything.

`function` is the function's name. It's followed by a pair of parentheses, which can, optionally, contain values passed to the function. These values are called *arguments*. Not every function features arguments. Then come the curly brackets and any statements that help the function do its thing.

Functions that return a value must use the `return` keyword. The `return` statement either ends the function directly or passes a value back to the statement that called the function. For example:

```
return;
```

This statement ends a function and does not pass on a value. Any statements in the function after `return` are ignored.

```
return(something);
```

This statement passes the value of the *something* variable back to the statement that called the function. The *something* must be of the same variable type as the function, an `int`, the `float`, and so on.

Functions that don't return values are declared of the `void` type. Those functions end with the last statement held in the curly brackets; a `return` statement isn't required.

One more important thing! Functions must be *prototyped* in your code. That's so that the compiler understands the function and sees to it that you use it properly. The prototype describes the value returned and any values sent to the function. The prototype can appear as a statement at the top of your source code. Listing 10-1 shows an example at Line 3.

Listing 10-1: Basic Function; No Return

```
#include <stdio.h>

void prompt();      /* function prototype */

int main()
{
    int loop;
    char input[32];

    loop=0;
    while(loop<5)
    {
        prompt();
        fgets(input,31,stdin);
        loop=loop+1;
    }
    return(0);
}

/* Display prompt */

void prompt()
{
    printf("C:\\DOS> ");
}
```

Exercise 10-1: Use the source code from Listing 10-1 to create a new project, ex1001. Build and run.

The program displays a prompt five times, allowing you to type various commands. Of course, nothing happens when you type, although you can program those actions later, if you like. Here's how this program works in regard to creating a function:

Line 3 lists the function prototype. It's essentially a copy of the first line of the function (from Line 22), but ending with a semicolon. It can also be written like this:

```
void prompt(void);
```

Because the function doesn't require any arguments (the items in parentheses), you can use the `void` keyword in there as well.

Line 13 accesses the function. The function is called as its own statement. It doesn't require any arguments or return any values, and it appears on a line by itself, as shown in the listing. When the program encounters that statement, program execution jumps up to the function. The function's statements

are executed, and then control returns to the next line in the code after the function was called.

Lines 22 through 25 define the function itself. The function type is specified on Line 22, followed by the function name, and then the parentheses. As with the prototype, you can specify `void` in the parentheses because no argument is passed to the function.

The function's sole statement is held between curly brackets. The `prompt()` function merely outputs a prompt by using the `printf()` function, which makes it seem like the function isn't necessary, but many examples of one-line functions can be found in lots of programs.

Exercise 10-2: Modify the source code from Listing 10-1 so that the `while` loop appears in its own function. (Copy Lines 7 through 16 into a new function.) Name that function `busy()` and have the `main()` function call it.

- C has no limit on what you can do in a function. Any statements you can stuff into the `main()` function can go into any function. Indeed, `main()` is simply another function in your program, albeit the program's chief function.
- When declaring an `int` or `char` function type, you can also specify `signed`, `unsigned`, `long`, and `short`, as appropriate.

- The `main()` function has arguments, so don't be tempted to edit its empty parentheses and stick the word `void` in there. In other words, this construct is wrong:

```
int main(void)
```

  The `main()` function in C has two arguments. It's possible to avoid listing them when you're not going to use them, by keeping parentheses empty. Chapter 15 discusses using the `main()` function's arguments.

- Other programming languages may refer to a function as a *subroutine* or *procedure*.

Prototyping (or not)

What happens when you don't prototype? As with anything in programming, when you goof up, the compiler or linker lets you know with an error message — or the program just doesn't run properly. It's not the end of the world — no, not like programming a military robot or designing genetic code for a new species of Venus flytrap.

Exercise 10-3: Modify the source code from Exercise 10-1. Comment out the prototype from Line 3. Build the result.

Compiler errors are wonderful things, delightfully accurate yet entirely cryptic. Here is the error message generated by Code::Blocks, although I list only the relevant parts of the message:

```
13 Warning: implicit declaration of function 'prompt'
23 Warning: conflicting types for 'prompt'
13 Warning: previous implicit declaration of 'prompt' was
          here
```

The first warning occurs at Line 13 in my source code file, where the `prompt()` function is used inside the `main()` function. The compiler is telling you that you're using a function without a prototype. As the error message says, you're implicitly declaring a function. That's a no-no, but not a full-on error.

The second warning occurs where the `prompt()` function dwells in the program. In my source code, it's at Line 23. The warning states that `prompt()` was already declared (at Line 11) and that the second use may conflict with the first.

The final warning is a reference back to where the function was called, again at Line 13.

To put it succinctly: The compiler has no idea what's up with the `prompt()` function. Your code compiles, but running it is risky.

You may draw the conclusion that prototyping is an absolute necessity in your C code. That's not entirely true. You can avoid prototyping by reordering the functions in your source code. As long as a function is listed before it's used, you don't need a prototype.

Exercise 10-4: Edit your source code from Exercise 10-3. Remove the function prototype that was commented out at Line 3. Cut and paste (move) the `prompt()` function from the bottom of the source code listing to the top, above the `main()` function. Save, build, and run.

Listing 10-2 shows what I conjured up as a solution for Exercise 10-4.

Listing 10-2: Avoiding the Function Prototype

```
#include <stdio.h>

/* Display prompt */

void prompt(void)
{
    printf("C:\\DOS> ");
}

int main()
{
    int loop;
    char input[32];

    loop=0;
    while(loop<5)
    {
        prompt();
        fgets(input,31,stdin);
        loop=loop+1;
    }
    return(0);
}
```

In this book, as well as in my own programs, I write the `main()` function first, followed by other functions. I find that this method allows for better readability, although you're free to put your own functions first to avoid prototyping. And if you don't, keep in mind that other programmers may do it that way, so don't be surprised when you see it.

Compiler error messages in Code::Blocks have parentheses after them. The parenthetical comments refer to the *switch*, or traditional command-line option, that enables checking for a particular warning. For example, the error messages from Exercise 10-3 read in full:

```
11 Warning: implicit declaration of function 'prompt'
          (-Wimplicit-function-declaration)
20 Warning: conflicting types for 'prompt' (enabled by
          default)
```

I don't list those items in this section because they junk up the way the text presents itself, as you can see in the preceding example.

Functions and Variables

I'm fond of saying that functions gotta funct. That is, they need to do something, to work as a machine that somehow manipulates input or generates

output. To make that happen, you need to know how to employ variables to, from, and within a function.

Using variables in functions

Functions that use variables must declare those variables — just like the `main()` function does. In fact, it's pretty much the same thing. The big difference, which you need to remember, is that variables declared and used within a function are local to that function. Or, to put it in the vernacular, what happens in a function stays within the function. See Listing 10-3.

Listing 10-3: Local Variables in a Function

```
#include <stdio.h>

void vegas(void);

int main()
{
    int a;

    a = 365;
    printf("In the main function, a=%d\n",a);
    vegas();
    printf("In the main function, a=%d\n",a);
    return(0);
}

void vegas(void)
{
    int a;

    a = -10;
    printf("In the vegas function, a=%d\n",a);
}
```

Both the `main()` and `vegas()` functions declare an `int` variable *a*. The variable is assigned the value `365` in `main()` at Line 9. In the `vegas()` function, variable *a* is assigned the value `-10` at Line 20. Can you predict the program's output for the `printf()` function on Line 12?

Exercise 10-5: Create a new project using the source code from Listing 10-3. Build and run.

Here's the output I see:

```
In the main function, a=365
In the vegas function, a=-10
In the main function, a=365
```

Even though the same variable name is used in both functions, it holds a different value. That's because variables in C are local to their functions: One function cannot change the value of a variable in another function, even if both variables sport the same type and name.

- My admonition earlier in this book about not duplicating variable names doesn't hold for variables in other functions. You could have 16 functions in your code, and each function uses the *alpha* variable. That's perfectly okay. Even so:
- You don't have to use the same variable names in all functions. The `vegas()` function from Listing 10-3 could have declared its variable as *pip* or *wambooli*.
- To allow multiple functions to share a variable, you specify a global variable. That topic is avoided until Chapter 16.

Sending a value to a function

The key way to make a function funct is to give it something to chew on — some data. The process is referred to as *passing an argument to a function,* where the term *argument* is used in C programming to refer to an option or a value. It comes from the mathematical term for variables in a function, so no bickering is anticipated.

Arguments are specified in the function's parentheses. An example is the `puts()` function, which accepts text as an argument, as in

```
puts("You probably shouldn't have chosen that option.");
```

The `fgets()` function swallows three arguments at once:

```
fgets(buffer,27,stdio);
```

Arguments can be variables or immediate values, and multiple arguments are separated by commas. The number and type of values that a function requires must be specified when the function is written and for its prototype as well. Listing 10-4 illustrates an example.

Listing 10-4: Passing a Value to a Function

```
#include <stdio.h>

void graph(int count);

int main()
{
```

```
    int value;

    value = 2;

    while(value<=64)
    {
        graph(value);
        printf("Value is %d\n",value);
        value = value * 2;
    }
    return(0);
}

void graph(int count)
{
    int x;

    for(x=0;x<count;x=x+1)
        putchar('*');
    putchar('\n');
}
```

When a function consumes an argument, you must clearly tell the compiler what type of argument is required. In Listing 10-4, both the prototype at Line 3 and the `graph()` function's definition at Line 20 state that the argument must be an `int`. The variable *`count`* is used as the `int` argument, which then serves as the variable's name inside the function.

The `graph()` function is called in Line 13, in the midst of the `while` loop. It's called using the *`value`* variable. That's okay; the variable you pass to a function doesn't have to match the variable name used inside the function. Only the variable type must match, and both `count` and `value` are `int` types.

The `graph()` function, from Line 20 through Line 27, displays a row of asterisks. The length of the row (in characters) is determined by the value sent to the function.

Exercise 10-6: Fire up a new project using the source code from Listing 10-4. Save the project as ex1006. Build it. Can you guess what the output might look like before running it?

Functions don't necessarily need to consume variables. The `graph()` function from Listing 10-4 can gobble any `int` value, including an immediate value or a constant.

Exercise 10-7: Edit the source code from Exercise 10-6, changing Line 13 so that the `graph()` function is passed a constant value of `64`. Build and run.

It's possible to pass a string to a function, but until you've read Chapter 12 on arrays and especially Chapter 18 on pointers, I don't recommend it. A string is really an array, and it requires special C language magic to pass the array to a function.

Sending multiple values to a function

C offers no limit on how many arguments a function can handle. As long as you properly declare the arguments as specific types and separate them all with commas, you can stack 'em up like commuters on a morning train, similar to this prototype:

```
void railway(int engine, int boxcar, int caboose);
```

In the preceding line, the `railway()` function is prototyped. It requires three `int` arguments: `engine`, `boxcar`, and `caboose`. The function must be passed three arguments, as shown in the prototype.

Exercise: 10-8: Modify the source code from Listing 10-4 so that the `graph()` function accepts two arguments; the second is the character to display.

Creating functions that return values

A great majority of the C language functions return a value; that is, they generate something. Your code may not use the values, but they're returned anyway. For example, both `putchar()` and `printf()` return values, and I've never seen a single program use those values.

Listing 10-5 illustrates a function that is sent a value and then returns another value. That's the way most functions work, although some functions return values without necessarily receiving any. For example, `getchar()` returns input but doesn't require any arguments. In Listing 10-6, the `convert()` function accepts a Fahrenheit value and returns its Celsius equivalent.

Listing 10-5: A Function That Returns a Value

```
#include <stdio.h>

float convert(float f);

int main()
{
    float temp_f,temp_c;
```

```
    printf("Temperature in Fahrenheit: ");
    scanf("%f",&temp_f);
    temp_c = convert(temp_f);
    printf("%.1fF is %.1fC\n",temp_f,temp_c);
    return(0);
}

float convert(float f)
{
    float t;

    t = (f - 32) / 1.8;
    return(t);
}
```

Line 3 in Listing 10-5 declares the `convert()` function's prototype. The function requires a floating-point value and returns a floating-point value.

The `convert()` function is called in Line 11. Its return value is stored in variable *`temp_c`* on that same line. In Line 12, `printf()` displays the original value and the conversion. The `.1f` placeholder is used. It limits floating-point output to all numbers to the left of the decimal, but only one number to the right. (See Chapter 13 for a full description of the `printf()` function's placeholders.)

The `convert()` function begins at Line 16. It uses two variables: *`f`* contains the value passed to the function, a temperature in Fahrenheit. A local variable, *`t`*, is used to calculate the Celsius temperature value, declared at Line 18 and assigned by the formula on Line 20.

Line 20 converts the `f` Fahrenheit value into the `t` Celsius value. The parentheses surrounding `f - 32` direct the compiler to perform that part of the calculation first and then divide the result by 1.8. If you omit the parentheses, 32 is divided by 1.8 first, which leads to an incorrect result. See Chapter 11 for information on the order of precedence, which describes how C prefers to do long math equations.

The function's result is sent back in Line 21 by using the `return` keyword.

Exercise 10-9: Type the source code from Listing 10-5 into your editor. Build and run.

Functions that return values can have that value stored in a variable, as shown in Line 11 of Listing 10-5, or you can also use the value immediately. For example:

```
printf("%.1fF is %.1fC\n",temp_f,convert(temp_f));
```

Exercise 10-10: Edit the source code from Listing 10-5 so that the `convert()` function is used immediately in the `printf()` function. ***Hint:*** That's not the only line you need to fix up to make the change complete.

You may also notice that the `convert()` function itself has a redundant item. Do you really need the *t* variable in that function?

Exercise 10-11: Edit your source code from Exercise 10-10 again, this time paring out the *t* variable from the `convert()` function.

Honestly, you could simply eliminate the `convert()` function altogether because it's only one line. Still, the benefit of a function like that one is that you can call it from anywhere in your code. So rather than repeat the same thing over and over, and have to edit that repeated chunk of text over and over when something changes, you simply create a function. Such a thing is perfectly legitimate, and it's done all the time in C.

And just because I'm a good guy, but also because it's referenced earlier in this chapter, Listing 10-6 shows my final result for Exercise 10-11.

Listing 10-6: A Tighter Version of Listing 10-5

```
#include <stdio.h>

float convert(float f);

int main()
{
    float temp_f;

    printf("Temperature in Fahrenheit: ");
    scanf("%f",&temp_f);
    printf("%.1fF is %.1fC\n",temp_f,convert(temp_f));
    return(0);
}

float convert(float f)
{
    return(f - 32) / 1.8;
}
```

The `convert()` function's math is compressed to one line, so a temporary storage variable (*t* from Line 18 in Listing 10-5) isn't needed.

Returning early

The `return` keyword can blast out of a function at any time, sending execution back to the statement that called the function. Or, in the case of the `main()` function, `return` exits the program. That rule holds fast even when `return` doesn't pass back a value, which is true for any `void` function you create. Consider Listing 10-7.

Listing 10-7: Exiting a Function with return

```
#include <stdio.h>

void limit(int stop);

int main()
{
    int s;

    printf("Enter a stopping value (0-100): ");
    scanf("%d",&s);
    limit(s);
    return(0);
}

void limit(int stop)
{
    int x;

    for(x=0;x<=100;x=x+1)
    {
        printf("%d ",x);
        if(x==stop)
        {
            puts("You won!");
            return;
        }
    }
    puts("I won!");
}
```

The silly source code shown in Listing 10-7 calls a function, `limit()`, with a specific value that's read in Line 10. A loop in that function spews out numbers. If a match is made with the function's argument, a `return` statement (refer to Line 25) bails out of the function. Otherwise, execution continues and the function simply ends. No `return` function is required at the end of the function because no value is returned.

Exercise 10-12: Create a new project using the source code shown in Listing 10-7. Build and run.

One problem with the code is that it doesn't check to ensure that only values from 0 to 100 are input.

Exercise 10-13: Modify the source code from Listing 10-7 so that a second function, `verify()`, checks to confirm whether the value input is within the range from 0 to 100. The function should return the constant TRUE (defined as `1`) if the value is within the range, or FALSE (defined as `0`) if not. When a value is out of range, the program needs to display an error message.

Of course, you always win after you've confined input for Exercise 10-13 to the given range. Perhaps you can figure out another way to code the `limit()` function so that the computer has a chance — even if it cheats?